GUIDE TO CONSTRUCT 2'S ADVANCED EVENT FEATURES

9

Ashley Construct Team Founder
published on 13 Apr, 2012 at 15:47

245 favourites

Tagged

Advanced
construct2
workflow

Stats

43,820 visits,
92,122 views

Tools

Construct 2 offers some advanced features in the event sheet. These can allow expert users to make the most of the event system, enabling more sophisticated logic than normally possible with standard events. This tutorial summarises the event features that are intended for advanced use or work differently to normal events, with some tips and tricks.

This tutorial is intended for experienced users. Be sure to read How Events Work if you haven't already, as this tutorial builds on those basics.

Translations

French
Guide de construire de deux fonctions d'événements avancés
TabloidA

Portugese (Brazillian)
Guide to Construct 2's recursos para eventos avançados
Ed0Andrew

Russian
Расширенные функции листа событий Construct 2
flameneon

Spanish
Guia avanzada de las características de los eventos para Construct 2.
katzin

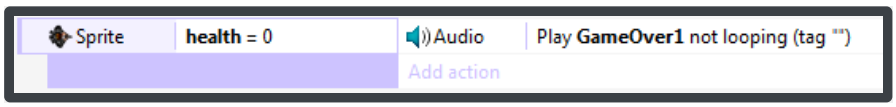
CREATE A TRANSLATION

License

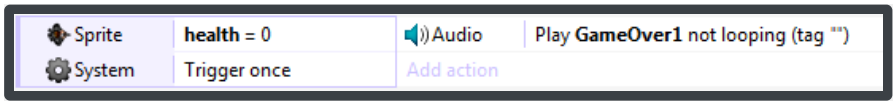
Trigger once while true

This condition effectively makes a continuously-true event in to a trigger. It tests "if did not reach this condition last tick". Usually it is placed as the last condition in an event, because if it is the first condition, it will be reached immediately and therefore never satisfy its test.

One of the places *Trigger once* is useful is for playing sounds. Consider the following event:



This is a common mistake - remember that the event is run every tick, which is about 60 times a second on most computers - and results in the sound being played and replayed continuously as long as the player's health variable equals 0. Often the result sounds awful. Instead, we want to play a single sound effect the first time the health counter equals 0. Adding *Trigger once* achieves this:



Now the sound will only play the first time *health* reaches 0. It will not play again until *health* changes to something else then goes back to 0, when it will play a single sound again.

A useful trick is to put *Trigger once* by itself in a group of events:

If the group is being enabled and disabled during the game, this will run the event once when the group is enabled. It acts as a sort of "On group enabled" trigger.

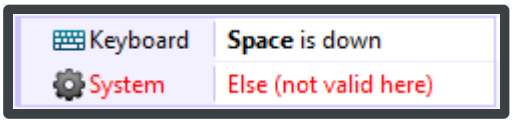
Pick all

As described in *How Events Work*, events typically work by *filtering* instances that don't meet the conditions. A subset of instances are left which meet all the conditions, then the actions run on those instances.

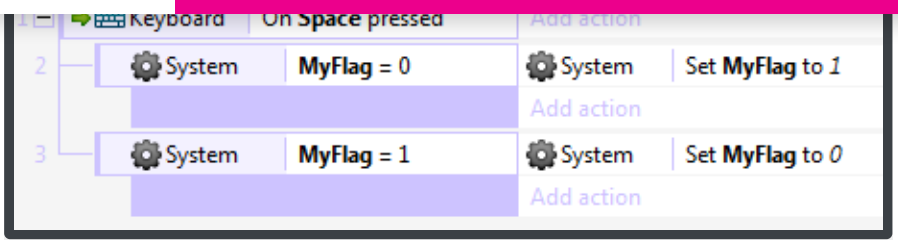
Pick all is the only condition which works in reverse: it restores all objects, so subsequent conditions pick from all instances again. It is difficult to come up with a simple intuitive example of this, but it can be useful for advanced users having to deal with deeply nested subevents where it is convenient to reset the picked objects.

Else

The *Else* event runs if the previous event did not run. It cannot be placed after a trigger, and must be the first condition in the event. If the *Else* is not in the right place, it will appear red indicating you must move it, as shown below. You cannot preview or export your project if *Else* conditions are in the wrong place, since the logic does not make sense.

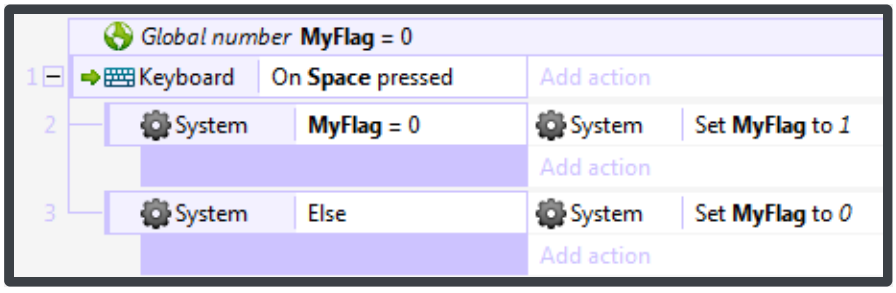


A common place *Else* is useful is when toggling a flag or variable. Often new users make the following



Notice how if *MyFlag* is 0 it is set to 1, but then the next event immediately sets it back to 0 again, because *MyFlag* is equal to 1! (This is where the order of events is important: remember events run top-to-bottom.) This event does not have the intended effect and *MyFlag* remains 0 no matter how many times Space is pressed.

One solution is to set *MyFlag* to **1 - MyFlag**. However, this only works for numbers, and is not always very readable. *Else* can solve the problem:

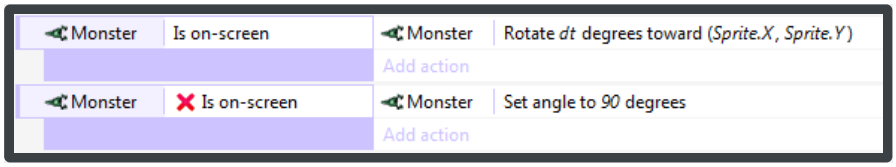


Now pressing Space correctly toggles *MyFlag* between 0 and 1, because the *Else* only runs if the previous event did not.

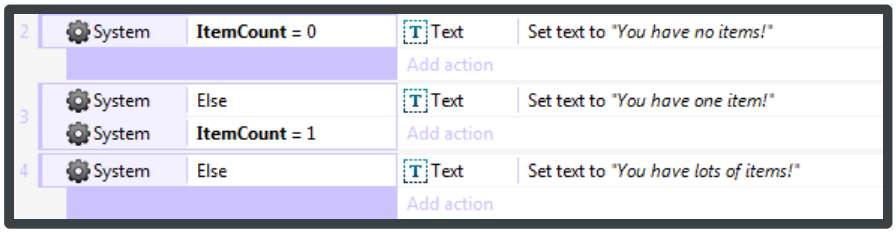
Pressing **X** is a keyboard shortcut to add an *Else* event after the currently selected event.

Note *Else* does **not** pick any objects. It literally just means "last event did not run". Consider the following example:

The intent may be to make monsters on-screen rotate toward the player, and make the rest point downwards at 90 degrees. There are two problems with this. Firstly, the *Else* event will not run at all if *any* monsters are on-screen, since then the first event has run so the *Else* does not run. Secondly, even if the *Else* event does run, it does not specifically pick monsters which are offscreen: it affects *all* monsters, because it does not pick instances. In this case, it is better to simply replace the *Else* with an inverted *Is on-screen* condition, as shown below. This will have the intended effect on instances on and offscreen.



Else can also be chained in to *"Else-if"* blocks by adding extra conditions to the *Else* event, and then adding another *Else* event after that. This is shown below.



This can be read:

If *ItemCount* is 0: set text to "You have no items!"

Else if *ItemCount* is 1: set text to "You have one item!"

WHILE LOOPS

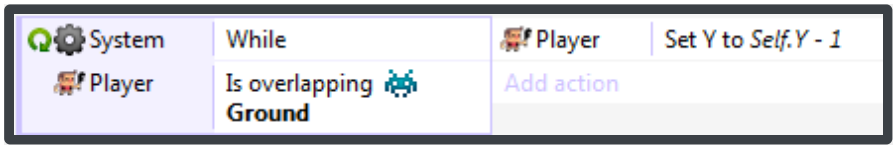
Another interesting system condition is the *While* loop. The other loops (repeat, for, for-each) are relatively straightforward, but *While* works slightly differently and so deserves its own mention.

While runs the event infinitely until either:

- a condition following it becomes false, or

- the *Stop loop* system action is used.

Most commonly it will be used with a condition following it, like so:

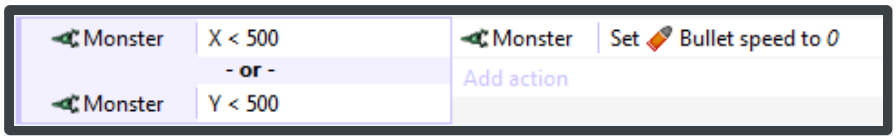


This will keep moving the player 1 pixel up until it no longer is overlapping 'Ground'. It happens instantly, so will repeat as many times as it needs to until 'Is overlapping Ground' becomes false.

Be careful not to accidentally create infinite loops. If the condition never becomes false during the loop, or the 'While' condition is used on its own without a 'Stop loop' action, it will hang while it loops forever.

'AND' BLOCKS VS. 'OR' BLOCKS

Normal events use 'And' logic: *all* conditions must be met for the actions to run. In other words, "condition 1 AND condition 2 AND condition 3..."

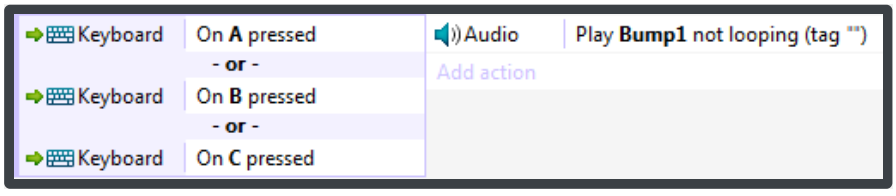


The intent here is to stop any monster which is left of X=500 *or* above Y=500. In this case, if any instances meet either condition, the event runs. The instances picked for the event will be those matching *either* condition.

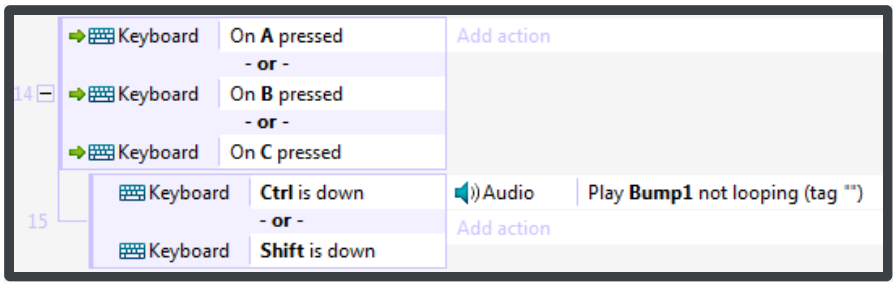
Note how in ordinary 'AND' blocks, subsequent conditions *filter out* instances not meeting the event - progressively *reducing* the number of picked instances. In contrast, 'OR' blocks *add* instances meeting the event - progressively *increasing* the number of picked instances.

By default event blocks are the normal 'And' type. They can be toggled to and from 'Or' blocks by right-clicking the event margin and selecting *Make 'Or' block*, or pressing the **Y** keyboard shortcut.

Normally, only one trigger can be placed in an event. However, this does not apply to 'Or' blocks. Multiple triggers can be added to an 'Or' event, and the actions will run when any of the triggers in the event fire. For example, below shows an event which plays a sound if either the A, B or C keys on the keyboard are pressed:



Using subevents, you can combine 'Or' blocks with 'And' blocks to create more advanced logic. For example:



This will play a sound on pressing A, B or C, so long as either Control or Shift are also held down. Alternatively: "If (A pressed OR B pressed OR C pressed) AND (Control is down OR Shift is down): play sound".

If the second event was not an 'Or' block, it would read "Control is down AND shift is down". Therefore, a sound would play if you hold down Control + Shift and press A, B or C. Alternatively: "If (A pressed OR B pressed OR C pressed) AND (Control is down AND Shift is down): play sound". So using subevents can help you make more advanced conditional logic.

OTHER USEFUL ADVANCED FEATURES

Unique IDs (UIDs) and Instance IDs (IIDs)

UIDs and IIDs are often useful to advanced users for advanced instance picking.

A UID can be used as a "reference" to an object. A UID can be stored in a variable and the object later

second Sprite's X co-ordinate (since Construct 2 uses zero-based indices). For more information see *Object expressions* in the manual entry on expressions. IIDs can also be used to pick instances using the *Pick Nth instance* system condition, but beware that an IID is not a permanent reference to an object: for that a UID should be used instead.

For more information, see UIDs and IIDs in the manual.

MOST ADVANCED OF ALL...

If you have previous programming experience or just want to take it to the next level, you can use the Javascript SDK to write your own plugins and behaviors for Construct 2. The overview page has a list of links to help you learn Javascript.

1 COMMENTS

Order by Best

Want to leave a comment? Login or Register an account!

[-] abdo html5 1 points 4 years ago

good tut

Permalink Reply